Comp 411
Principles of Programming Languages
Lecture 12
The Semantics of Recursion III & Loose Ends

Corky Cartwright

February 6, 2023

Call-by-name vs. Call-by-value Fixed-Points

Given a recursive definition $f \overset{m}{=} E_f$ in a call-by-value language where E_f is an expression constructed from constants in the base language and `f`. What does it mean?

Example: let `D` be the domain of Scheme values. Then the base operations are continuous call-by-value functions on `D` and

```
fact := map n to if n = 0 then 1 else n * fact(n-1)
```

is a recursive definition of a function on `D`.

In a *call-by-name* language `map n to ...` is interpreted using call-by-name β-reduction, the meaning of `fact` is

$Y(\text{map fact to } E_{fact})$

What if `map` (`λ`-abstraction) has *call-by-value* semantics? `Y` does not quite work because evaluations of form $Y(\text{map } f \text{ to } E_f)$ diverge with call-by-value β-reduction.

Defining `Y` in a Call-by-value Language

We want to define $\mathtt{Y}_{\mathtt{v}}$, a call-by-value variant of `Y`.

Key trick: use η(eta)-conversion to delay the evaluation of `F(x x)` inside of the expression defining `Y`. In the mathematical literature on the λ-calculus, η-conversion is often assumed as an axiom. In models of the pure λ-calculus, it typically holds.

Definition: η-conversion is the following equation:

```
M = λx . Mx
```

where `x` is not free in `M`. *If* the λ-abstraction used in the definition of `Y` has call-by-value semantics, then given the functional `F` corresponding to recursive function definition, the computation `YF` diverges. We can prevent this from happening by η-converting both occurrences of `F(x x)` within `Y`.

What Is the Code for $\mathbf{Y_v}$?

- `Yv = λF. (λx.(λy.(F(x x))y)) (λx.(λy.(F(x x))y))`
- Does this work for Scheme (or Java with an appropriate encoding of functions as anonymous inner classes) where **λ**-binding has call-by-value semantics? Yes!
- Let **G** be some functional `λf.λn.M`, like `FACT`, for a *unary* recursive *function definition*. **G** and `λn.M` are values (**λ**-abstractions). Then

```
Yv G = (λx.(λy.(G(x x)) y)) (λx.(λy.(G(x x)) y))
     = λy.[G((λx.(λy.(G(x x)) y)) (λx.(λy.(G(x x)) y))) y]
     = G((λx.(λy.(G(x x)) y)) (λx.(λy.(G(x x)) y)))
```

is a *value*. In call-by-value, `Y G` is *not* a value but `Yv G` is.

- But `G(Yv G) = (λf.λn.M)(Yv (λf.λn.M)) = λn.M[f:=Yv(λf.λn.M)]`, which is a *value*.
- As shown above (using call-by-value β-conversion) `YvG = G(YvG)` where **G** is any closed functional `λf.λn.M`.
- Disadvantage of $\mathbf{Y_v}$ vs. **Y**: $\mathbf{Y_v}$ is arity-specific for recursive function definitions in languages like Jam that support multiple arguments in **λ**-abstractions. (Note: unary $\mathbf{Y_v}$ works for all curried function definitions since every **λ**-abstraction is unary.) b

Alternate Definitions of Y_v

- The following defintion of the call-by-value version **Y** also works:
  Y_v `= λF. (λx. F(λy.(x x)y)) (λx. F(λy.(x x)y))`
- In this case, we **η**-convert `(x x)` instead of `F(x x)`.
- Let **G** be some functional `λf.λn.M`, like `FACT`, for a *unary* recursive *function definition*. **G** and `λn.M` are values (**λ**-abstractions). Since **G** has the form `λf.λn.M`

  Y_v `G = (λx. G(λy.(x x)y)) (λx. G(λy.(x x)y)))`
  `= G(λy. (λx. G(λy.(x x)y)) (λx. G(λy.(x x)y)))`
  `= λn.M[f := λy. (λx. G(λy.(x x)y)) (λx. G(λy.(x x)y))`

  which is a value in both call-by-value and call-by-name.

  In call-by-value, `Y G` is *not* a value but Y_v `G` is.
- But `G(`Y_v`G) = (λf.λn.M)(`Y_v` (λf.λn.M)) = λn.M[f:=`Y_v`(λf.λn.M)]`, which is a *value*.
- As shown above (using call-by-value β-conversion) Y_v`G = G(`Y_v`G)` where **G** is any closed functional `λf.λn.M`.
- Disadvantage of Y_v vs. **Y**: Y_v is arity-specific for recursive function definitions in languages like Jam that support multiple arguments in **λ**-abstractions. (Note: unary Y_v works for all curried function definitions since every **λ**-abstraction is unary.)

Loose Ends

- Meta-errors
- Read the notes!
- `letrec` (in notes)

Lazy JamVal: a Concrete Example

Consider Jam with call-by-value **λ** and lazy `cons`. What is the domain `JamVal` of data values? It consists of the flat domain of integers $\mathbf{Z}_\perp$ augmented by `JamList`, the domain of lazy lists over `JamVal`s, and the function domain $\texttt{JamVal}^k \rightarrow \texttt{JamVal}$ of call-by-value functions of arity $\mathbf{k}$ for $\mathbf{k} \in \mathbb{N}$ (natural numbers).

$$\texttt{JamVal} = \mathbf{Z}_\perp + \texttt{JamList} + \bigcup_k \texttt{JamVal}^k \rightarrow \texttt{JamVal}$$

$$\texttt{JamList} = \texttt{JamEmpty} + \texttt{cons(JamVal, JamList)}$$

where `cons` is lazy (non-strict) in both arguments. Does call-by-value $\mathbf{Y}_\mathbf{v}$ let us recursively define infinite trees? Yes!

Call-by-value **`Y`** with Lazy Lists

Assume we want to define the infinite lazy tree with no leaves:

```
consMax = cons(consMax, consMax)
```

How do we express this in Jam? We need **`letrec`** (**`let`** with recursive binding):

```
letrec consMax := cons(consMax,consMax);
in consMax
```

What is the denotational meaning of recursive definition? The least call-by-value fixed-point (using $\mathbf{Y_v}$) of the corresponding function **`C`** which is **`λc.cons(c,c)`**. Since **`cons`** is lazy, the standard least fixed point construction yields the desired infinite tree. Try evaluating $\mathbf{Y_v}$ **`C`** in the Assignment 3 reference interpreter (using *value-need* mode).